Universidad de Valladolid

ESCUELA TÉCNICA SUPERIOR DE INGENIEROS DE TELECOMUNICACIÓN

## TRABAJO DE FIN DE MASTER

MASTER UNIVERSITARIO EN INVESTIGACIÓN EN TECNOLOGÍAS DE LA INFORMACIÓN Y LAS COMUNICACIONES

---

# Compresión de datasets RDF en HDT usando Spark

---

Autor: Carlos Barrales Ruiz
Tutor: Pablo de la Fuente Redondo
Miguel Ángel Martínez Prieto

## Abstract

Apache Spark is a general purpose big data processing framework using the mapreduce paradigm, quickly becoming very popular. Although the information provided by Spark authors indicates a substantial improvement in performance against Hadoop, there is very little evidence in the literature of specific tests that reliably proves such claims. In this Master Work study the benefits of Spark and the most important factors on which they depend, considering as a reference the transformation of RDF datasets into HDT format. The main objective of this work is to perform one exploratory study to leverage Spark solving the HDT serialization problem, finding ways to remove limitations of the current implementations, like the memory need which use to increase with the dataset size. To do that, first we've setup a open environment to ensure reproducibility and contributed with 3 different approaches implementing the most heavy task in the HDT serialization. The test performed with different dataset sizes showed the benefits obtained with the proposed solution compared to legacy Hadoop MapReduce implementation, as well as some highlights to improve even more the serialization algorithm.

## Keywords

semantic web, RDF, compression, HDT, parallelization, map-reduce, big data.

## Resumen

Apache Spark es un marco de desarrollo de procesamiento de datos de propósito general que utiliza el paradigma map-reduce muy popular en la actualidad. Aunque la información proporcionada por los autores de Spark indica una mejora sustancial en el rendimiento frente a Hadoop, hay muy poca evidencia en la literatura de pruebas específicas que demuestre totalmente tales afirmaciones. En este trabajo maestro se estudian los beneficios de Spark y los factores más importantes de los que dependen, considerando como referencia la transformación de conjuntos de datos RDF en formato HDT. El objetivo principal de este trabajo es realizar un estudio exploratorio para aprovechar Spark resolver el problema de serialización HDT, encontrando maneras de eliminar las limitaciones de las implementaciones actuales, como la necesidad de memoria que se utilizan para aumentar con el tamaño del conjunto de datos. Para ello, primero hemos configurado un entorno abierto para asegurar la reproducibilidad y hemos contribuido con la implementación de 3 algoritmos diferentes para resolver la tarea más pesada en la serialización HDT. La prueba realizada con diferentes tamaños de conjunto de datos muestra los beneficios obtenidos con la solución propuesta en comparación con la implementación de Hadoop MapReduce, así como algunos aspectos destacados para mejorar aún más el algoritmo de serialización.

## Palabras clave

web semántica, RDF, compresión, HDT, parallelización, map-reduce, big data.

## Acknowledgements

Quisiera agradecer brevemente a todos los que me han apoyado y animado a continuar con mi carrera, especialmente familia y amigos. A mi mujer Rebeca y a mi hija Claudia, lamentando el tiempo que no he podido disfrutar de ellas. También agradecer la labor de los profesores del máster, por su implicación, su consejo y su buen hacer. A mis tutores de TFM Pablo y Miguel Ángel por su ayuda, comprensión y guía. Gracias de corazón.

# Contents

# List of Figures

# List of Tables

# Part I.

# Introduction and previous knowledge

# 1

# Introduction

The Semantic Web isn't just about putting data on the web. It is about making links, so that a person or machine can explore the web of data. In this context, the term Linked Data refers to a set of best practices for publishing and connecting structured data on the Web so they are more related and hence more useful. The idea is simple: If we start publishing machine readable data in the web and we can link them, we will be building a big knowledge network which can be processed by machines. The Linked Data concept was proposed originally by Tim Berners-Lee and his 2006 Linked Data Principles [Berners-Lee, 2006], which is considered to be the official and formal introduction of the concept itself. At its current stage, Linked Data is a W3C-backed movement that focus on connecting data sets across the Web, and it can be viewed as a subset of the Semantic Web concept, which is all about adding meanings to the Web.[Yu, 2014a].

From a technical perspective, Linked Data is related to web published data for which the meaning is defined explicitly and linked to external data sets, which can be liked to other data sets as well (also external). Conceptually, refers to a good practices set for the publication and interconnection of structured data in the Web [Yu, 2014a]. The initiative is founded on the semantic web concept and recommendation of open data interchange standards (URI, HTTP,. . . ) and information model (RDF, XML, . . . ), being RDF [W3C, 2014b] one of the most popular data model, resource description and relations between them.

The Resouce Description Framework (RDF) was created b the W3C[1] in 1999 as one standard for metadata coding [Yu, 2014b]. Before going into details, it's convenient to detail the big picture of the abstract model in a high scale. RDF uses the abstract model to decompose the information or knowledge in small pieces, with some simple rules over the semantic (meaning) of each piece. The objective is to provide a general method that is simple and flexible enough to express any fact, but in a structured way so that the computer applications can operate with that knowledge [Yu, 2014b].

The RDF data model defines two key data structures [W3C, 2014b]:

- **RDF graphs** They are sets of triplets $\{subject, predicate, objetc\}$ where the el-

---

[1] http://www.w3c.org

ements can be IRIs[2], empty nodes or literal data types. They articulates the resource's descriptions.

- **RDF datasets** They are collection of RDF graphs and y que comprise a default graph and zero or more labeled graphs.

RDF also provides a specification of the SPARQL query language [3], which allows to perform search over RDF graph as well as manipulation operations. SPARQL was standardized by the *W3C's SPARQL Working Group* in January 2008 [Yu, 2014c] containing three specifications:

- SPARQL Query Language for RDF. The language specification itself.
- SPARQL Query Results XML Format. To describe query results and leverage other tools.
- SPARQL Data access protocol for remotely querying RDF databases.

Berners-Lee [Berners-Lee, 2006] outlined a set of "rules" for publishing data on the web in a way that all published data becomes part of a single global data space: Use URIs as names for things Use HTTP URIs so that people can look up those names When someone looks up a URI, provide useful information, using the standards (RDF, SPARQL). Include links to other URIs, so that they can discover more things.

The goal of the W3C Semantic Web Education and Outreach group's Linking Open Data community project is to extend the Web with data commons by publishing various RDF datasets on the Web and by setting RDF links between data items from different data sources [Wikimedia, nd]. In October 2007, datasets consisted of over two billion RDF triples, which were interlinked by over two million RDF links. By September 2011 this had grown to 31 billion RDF triples, interlinked by around 504 million RDF links. A detailed statistical breakdown was published in 2014 [Schmachtenberg et al., 2014].

Linked Data technologies are being using to share data covering a wide range of different topical domains. The table 1.1 below gives an overview of the topical domains of the 1014 datasets that were discovered by the crawl of [Schmachtenberg et al., 2014].

---

[2] Acronym of *Internationalized Resource Identifier*

[3] Acronym of SPARQL Protocol and RDF Query Language

| **Topic** | **Datasets** | **%** |
|---|---:|---:|
| Government | 183 | 18.05% |
| Publications | 96 | 9.47% |
| Life sciences | 83 | 8.19% |
| User-generated content | 48 | 4.73% |
| Cross-domain | 41 | 4.04% |
| Media | 22 | 2.17% |
| Geographic | 21 | 2.07% |
| Social web | 520 | 51.28% |
| Total | 1014 | |

Table 1.1.: Principal datasets topics at 2014.

The most visible example of adoption and application of the Linked Data principles has been the Linking Open Data project[4], a grassroots community effort founded in January 2007 and supported by the W3C Semantic Web Education and Outreach Group[5]. The original and ongoing aim of the project is to bootstrap the Web of Data by identifying existing data sets that are available under open licenses, converting these to RDF according to the Linked Data principles, and publishing them on the Web. Figure 1.1 shows the scale of Web of Data originating from the Linking Open Data project. Each node in this cloud diagram represents a distinct data set published as Linked Data, as of March 2009 [Bizer et al., 2009].

RDF format was designed with the goal of being document-oriented and readable. However, the need to exchange very large datasets has revealed the drawbacks of traditional RDF representations, including redundancy and high computational capacity required for processing. This scenario has motivated the appearance of meta-formats thought to represent large data sets efficiently under the points of view of the publication, exchange and retrieval of information. This the case of HDT. However, efficient coding of these types of formats, especially against large volumes of data, is a problem that remains open.

In summary, in the information age, the exploitation of large volumes of data in RDF format encoded in HDT format presents a great number of possibilities. A clear example of this is the growth of new data sources such as [W3C, 2016] strongly supported by standards such as [Pérez et al., 2006, Quilitz and Leser, 2008] and other semi-structured data technical specifications.

---

[4] `http://esw.w3.org/topic/SweoIG/TaskForces/CommunityProjects/LinkingOpenData` [Last check: Sep 2017]

[5] http://www.w3.org/2001/sw/sweo/

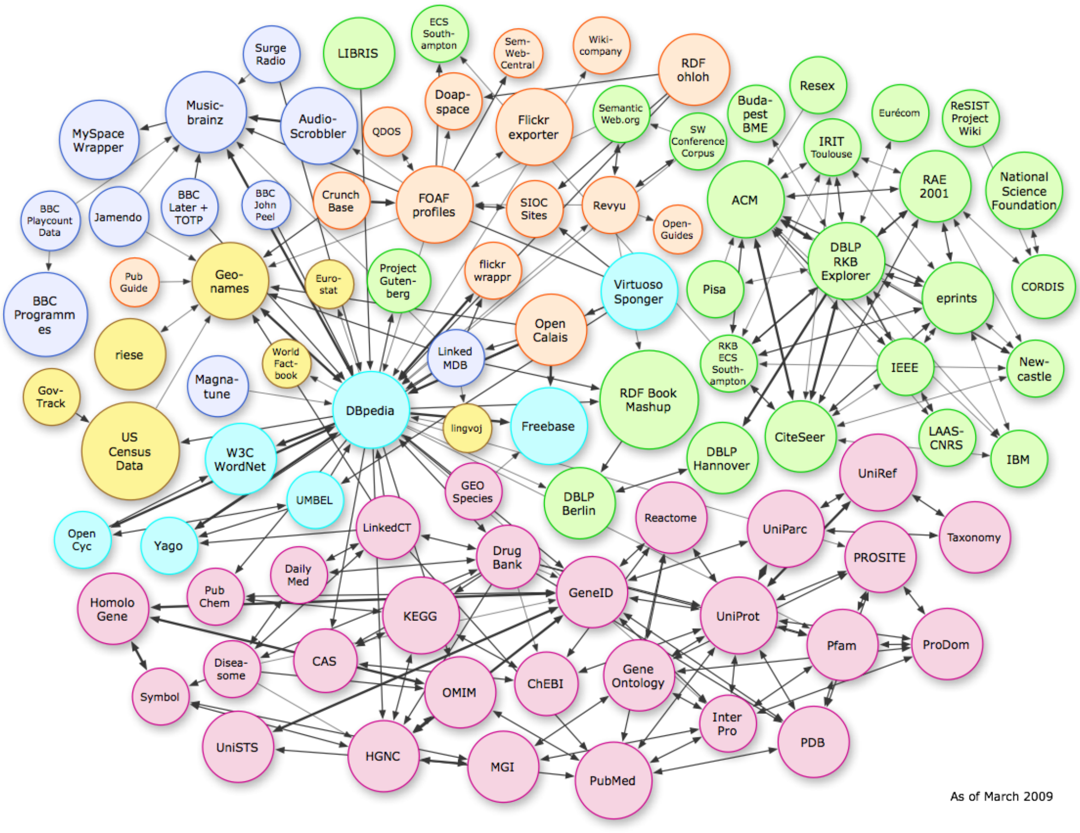


Figure 1.1.: Linking Open Data cloud diagram giving an overview of published data sets and their interlinkage relationships [Bizer et al., 2009].

## 1.1. Motivation and objectives

The main goal of this work is to discover and explore new ways to enhance the generation of HDT serializations of large RDF datasets, leveraging new generations of mapreduce frameworks and related technologies, avoiding memory limitations and disk usage slowdown distributing tasks.

Secondary objectives are:

- Study how to leverage the benefits coming with next generation of Hadoop Ecosystem Resouce Manager (YARN) and Apache Spark.
- Propose and setup a Open Framework using virtualization and/or Platform As a Service environment with all needed technologies to share and reproduce the experiment and set the foundation of future activities.
- Study the properties and restrictions of the HDT structures to address one approach able to build the serialization incrementally.
- Design and implement one distributed algorithm for Spark framework with the following considerations:
  - Data partitioning and aggregation strategy to reduce the memory consumption of current HDT-MR approach.
  - Re-design map and reduce tasks to lower the number of disk writes.
  - Discover or induce HDT specifications with better properties.

## 1.2. Document structure

The current document chapter provides a brief introduction to the problem and related concepts, standards and technologies. The whole document is organized as follows:

**First (introductory) part** :

**Chapter 1** Quick overview and goals.

**Chapter 2** Details the background about the main concepts.

**Chapter 3** Reviews history evolution of related technologies and methodology to carry out the objectives

**Second (technical detail) part** :

**Chapter 4** Explains the plan to validate and evaluate the outcomes.

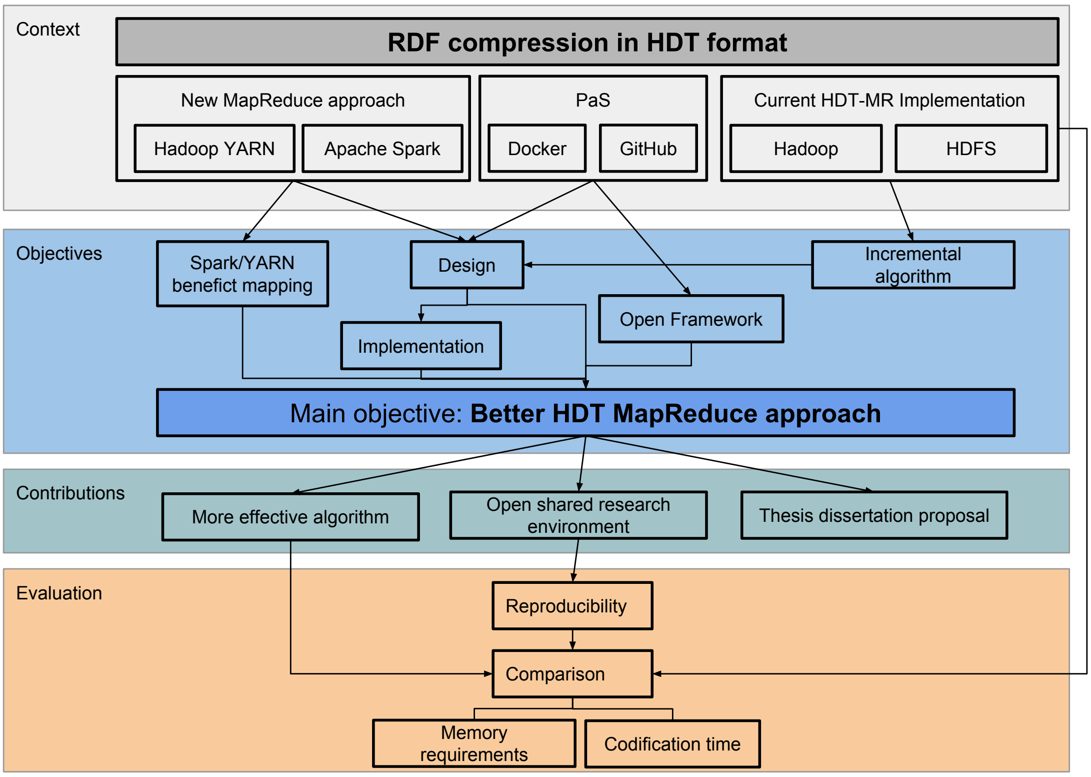


Figure 1.2.: TFM work schema

**Chapter 5** Details the technical solution performed.

**Chapter 6** Demonstration and discuss about the work outcomes.

**Third (conclusion) part** :

**Chapter 7** Explores related work and similar approaches to solve the same problem.

**Chapter 8** Raises this work conclusions.

**Chapter 9** Summary about the limitations of this works and short discussion about possible research line continuation.

# Part II.

# Previous knowledge

# 2

# RDF

The RDF model encodes data in the form of subject, predicate, object triples. The subject and object of a triple are both URIs that each identify a resource, or a URI and a string literal respectively. The predicate specifies how the subject and object are related, and is also represented by a URI [Bizer et al., 2009].

For instance, an RDF triple can state that two people, A and B, each identified by a URI, are related by the fact that A knows B. Similarly an RDF triple may relate a person C to a scientific article D in a bibliographic database by stating that C is the author of D. Two resources linked in this fashion can be drawn from different data sets on the Web, allowing data in one data source to be linked to that in another, thereby creating a Web of Data. Consequently it is possible to think of RDF triples that link items in different data sets as analogous to the hypertext links that tie together the Web of documents [Bizer et al., 2009].

The basic structure of the RDF data model is therefore a set of triples, each one composed by a subject, predicate and object [Yu, 2014b]:

**Subject** It's an IRI or a blank node and it's used to denote resources in the world. Must be identifdied by the URI. It's also named start node of an RDF graph.

**Object** It's es an IRI and it's used to denote resources in the world. It's also named end node of an RDF graph.

**Predicate** It's an IRI, a literal or a blank node. It's used to denote relaton between the subject and the object. It must be identified by the URI, also called edge in a RDF graph.

The set of such triples is called RDF graph. An RDF graph can be viewed as a node and the directed arc diagram, in which each triplet is represented as a $\{node, arc, node\}$ link. It is possible for an IRI predicate to also occur as a node in the same graph.

## 2.1. RDF Datasets

An RDF dataset is a collection of RDF graphs comprising:

- Exactly one predetermined graph.
- The default graph has no name and may be empty.
- Zero or more named graphs. Each named graph is a pair consisting of an IRI or a blank node (the name of the graph), and an RDF graph. Named graphs are unique within the data set.

## 2.2. RDF Representation

One of the most important aspects of RDF is that it is an abstract data model, and the RDF standard does not specify its representation. The recommended and perhaps most popular representation of an RDF model is the XML serialization format (known as RDF / XML). However, RDF / XML is not designed to be interpreted comfortably by a human, making it difficult to read and has a heavy syntax. For this reason, there are other RDF serialization formats such as Notation-3 (N3), TURTLE [1], N-Triples and N-Quads [W3C, 2014a, W3C, 2014c, W3C, 2014d, W3C, 2014e]. Since Notation-3 has some features that are not necessary for the serialization of RDF models (such as its support for rules based on RDF), TURTLE was created as a simplified subset of Notation-3 for this context. N-Triples is an even simpler simplification, so it became very popular nowadays. [Yu, 2014b].

Code 2.1: RDF/XML example

```
<rdf:RDF
xmlns="http://xmlns.com/foaf/0.1/"
xmlns:dc="http://purl.org/dc/terms/"
xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   >
<Document
rdf:about="http://www.w3.org/2001/sw/RDFCore/ntriples/">
<dc:title xml:lang="en-US">
N-Triples
</dc:title>
<maker>
<Person rdf:nodeID="art">
<name>Art Barstow</name>
</Person>
</maker>
<maker>
```

---

[1] Terse RDF Triple Language

```
<Person rdf:nodeID="dave">
<name>Dave Beckett</name>
</Person>
</maker>
</Document>
</rdf:RDF>
```

Code 2.2: Notation-3 example file

```
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix dc: <http://purl.org/dc/terms/> .

<http://www.w3.org/2001/sw/RDFCore/ntriples/>
a foaf:Document ;
dc:title "N-Triples"@en-US ;
foaf:maker [
a foaf:Person ;
foaf:name "Art Barstow"
], [
a foaf:Person ;
foaf:name "Dave Beckett"
] .
```

Code 2.3: Turtle sample

```
@base <http://example.org/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns
    #> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix rel: <http://www.perceive.net/schemas/
    relationship/> .

<#green-goblin>
rel:enemyOf <#spiderman> ;
a foaf:Person ;    # in the context of the Marvel
    universe
foaf:name "Green Goblin" .
```

Code 2.4: N-Triples sample

```
_:subject1 <http://an.example/predicate1> "object1" .
_:subject2 <http://an.example/predicate2> "object2" .
```

# 3

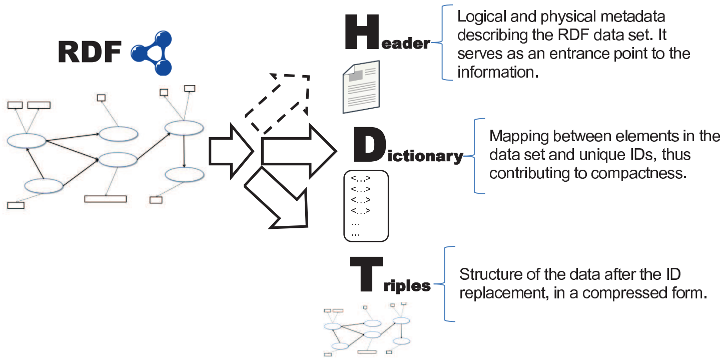
# HDT

HDT is a compact binary RDF representation proposed by [Fernández et al., 2013] to avoid the limitations in terms of resource utilization to store and process large RDF datasets, so the compact nature of HDT represents an improvement in storage space while leverages succint internal structures to speedup queries and lookups [Martínez-Prieto et al., 2012, Fernández et al., 2011]. HDT is composed of three main elements for which its name is an acronym:

- **Header** A header, including the metadata describing the RDF dataset that serves as the entry point to the information it contains.
- **Dictionary** A dictionary, the organization of all identifiers in the RDF graph. A catalog of the RDF terms (URI, blank nodes, literals) mentioned in the graphic with high levels of compression is provided.
- **Triples** A triple component, which comprises the pure structure of the underlying RDF, ie: compact the set of triples avoiding the ”noise” produced by long tags and repetitions common in the RDF format.

HDT provides two main advantages over RDF:

- It has a much smaller size.
- Its structure allows to be exploited efficiently by information retrieval technologies, such as SPARQL.



Logical and physical metadata describing the RDF data set. It serves as an entrance point to the information.

Mapping between elements in the data set and unique IDs, thus contributing to compactness.

Structure of the data after the ID replacement, in a compressed form.

Figure 3.1.: HDT Structure [Fernández et al., 2013]

The work of [Hernández Illera et al., 2015] details precisely how the HDT format is constructed and presents an improved implementation called HDT++, still under development.

# 4

# Problem description

## 4.1. State of the art

The research line in which this work is based is was started by J. D. Fernandez et al. and M. A. Martínez et al. between 2010 and 2011 [Fernández et al., 2010, Fernández et al., 2011]. They proposed HDT as one efficient serialization format to process RDF data. Subsequently, M. Arias et al. contributes in 2011 with his TFM entitled "Analysis of RDF datasets to improve compression using HDT" and a tool that demonstrates the practical applications of such format [Arias Gallego et al., 2011]. In 2012, Martínez et al. Presents HDT-FoQ, which improves the existing HDT specification by complementing it with indexes that allow more efficient queries. [Martínez-Prieto et al., 2012]. Fernandez presents his thesis titled "Binary RDF for Scalable Publishing, Exchanging and Consumption in the Web of Data" in which it contributes with an in-depth study of RDF data sets and proposes a compact data structure configuration to explore and query sets of Data encoded in HDT [Fernández et al., 2013]. Although within the theoretical framework HDT is a very advantageous representation, its construction requires large amounts of memory that increase the larger the volumes of data. This is why Giménez et. al contributed with HDT-MR presenting a method to serialize RDF data in a scalable HDT format using the MapReduce paradigm [Giménez, 2014].

The implementation of [Giménez, 2014] is based on one of the first specifications of Apache Hadoop 1.2. Unfortunately, Hadoop's 1.x versions have a strong reliance on file systems to distribute map and reduce tasks [Shi et al., 2015]. Moreover, although HDT-MR allows to increase productivity in HDT coding by distributing the load on a variable set of nodes, it still requires a large amount of memory since the data dictionary needs to be loaded completely in memory on all nodes. The limitations of the Hadoop 1.x branch have driven the need to improve the implementation of the paradigm to achieve lighter and more homogeneous solutions such as Apache YARN 2.0 (newer implementation of the Hadoop MR framework). This is the point at which this work starts. So given the basis and the initial clues ([Shi et al., 2015]), the question we raise is the following: Can we leverage current technology to improve the HDT serialization algorithm so we can be able to process arbitrary large sized RDF datasets?

It's well know that MapReduce is based on a master / slave architecture. The master initiates the process, distributes the workload between the cluster and manages all the information. The slaves (or workers) perform the mapping and reduction tasks. Workers commonly store data using a distributed file system based on the GFS (Google File System) model, where data is divided into small pieces and stored in different nodes. This allows workers to take advantage of local data as much as possible. MapReduce performs exhaustive input / output operations. Traditionally, the input of each task and each output is read and written to disk, although new implementations like Apache Spark allow the decision to persist in disk or in memory with flexibility. It is also intensive in the use of bandwidth. The output map must be transferred to reduce nodes and, although most map tasks read their data locally, some of them must be obtained from other nodes [Zaharia et al., 2012].

When a node has an empty task slot, Hadoop chooses a task for it from one of three categories. First, any failed tasks are given highest priority. This is done to detect when a task fails repeatedly due to a bug and stop the job. Second, non-running tasks are considered. For maps, tasks with data local to the node are chosen first. Finally, Hadoop looks for a task to execute speculatively. To select speculative tasks, Hadoop monitors task progress using a score between 0 and 1. For a map, the progress score is the fraction of input data read. For a reduce task, the execution is divided into three phases, each of which accounts for 1/3 of the score:

- The copy phase, when the task fetches map outputs.
- The sort phase, when map outputs a resorted by key.
- The reduce phase, when a user-defined function is applied to the list of map outputs with each key.

In each phase, the score is the fraction of data processed [Zaharia et al., 2008]

Hadoop's scheduler makes several implicit assumptions:

1. Nodes can perform work at roughly the same rate.
2. Tasks progress at a constant rate throughout time.
3. There is no cost to launching a speculative task on a node that would otherwise have an idle slot.
4. A task's progress score is representative of fraction of its total work that it has done. Specifically, in a reduce task, the copy, sort and reduce phases each take about 1/3 of the total time.

5. Tasks tend to finish in waves, so a task with a low progress score is likely a straggler.

6. Tasks in the same category (map or reduce) require roughly the same amount of work.

Given that, [Zaharia et al., 2008] proposes a new speculative task scheduler design by starting from first principles and adding features needed to behave well in a real environment. The primary insight behind our algorithm is as follows: We always speculatively execute the task that we think will finish farthest into the future, because this task provides the greatest opportunity for a speculative copy to overtake the original and reduce the job's response time. We explain how we estimate a task's finish time based on progress score below. That strategy is called LATE, for Longest Approximate Time to End. Intuitively, this greedy policy would be optimal if nodes ran at consistent speeds and if there was no cost to launching a speculative task on an otherwise idle node.

Along this line, in 2010 [Zaharia et al., 2010] presented a new cluster computing framework written in the Scala language called Spark, which supports applications with working sets while providing similar scalability and fault tolerance properties to MapReduce. Spark improves over Hadoop MapReduce, which helped ignite the big data revolution, in several key dimensions: it is much faster, much easier to use due to its rich APIs [Armbrust et al., 2015a, Zaharia et al., 2010]. The main abstraction in Spark is that of a resilient distributed dataset (RDD), which represents a read-only collection of objects partitioned across a set of machines that can be rebuilt if a partition is lost.

Users can explicitly cache an RDD in memory across machines and reuse it in multiple MapReduce-like parallel operations. RDDs achieve fault tolerance through a notion of lineage: if a partition of an RDD is lost, the RDD has enough information about how it was derived from other RDDs to be able to rebuild just that partition. Although RDDs are not a general shared memory abstraction, they represent a sweet-spot between expressivity on the one hand and scalability and reliability on the other hand, and we have found them well-suited for a variety of applications. Spark is the first system to allow an efficient, general-purpose programming language to be used interactively to process large datasets on a cluster.

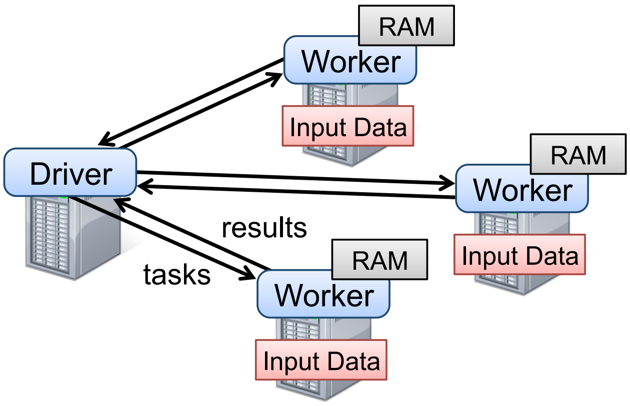


Figure 4.1.: Spark execution model [Zaharia et al., 2012]

There are three architectural components covering the majority of architectural differences between MapReduce and Spark [Shi et al., 2015]:

**Shuffle** The shuffle component is responsible for exchanging intermediate data between two computational stages. For example, in the case of MapReduce, data is shuffled between the map stage and the reduce stage for bulk synchronization. The shuffle component often affects the scalability of a framework. Very frequently, a sort operation is executed during the shuffle stage. An external sorting algorithm, such as merge sort, is often required to handle very large data that does not fit in main memory. Furthermore, aggregation and combine are often performed during a shuffle.

**Execution model** The execution model component determines how user defined functions are translated into a physical execution plan. The execution model often affects the resource utilization for parallel task execution. In particular:

1. Parallelism among tasks.
2. Overlap of computational stages.
3. data pipelining among computational stages.

**Caching** The caching component allows reuse of intermediate data across multiple stages to avoid recomputing them several times. Effective caching speeds up iterative algorithms at the cost of additional space in memory or on disk.

[Zaharia et al., 2012] evaluated Spark and RDDs through a series of experiments on Amazon EC2, as well as benchmarks of user applications. Overall, our results showed the following:

- Spark outperforms Hadoop by up to 20x in iterative machine learning and graph applications. The speedup comes from avoiding I/O and deserialization costs by storing data in memory as Java objects.
- Applications written by our users perform and scale well. In particular, we used Spark to speed up an analytics report that was running on Hadoop by 40x.
- When nodes fail, Spark can recover quickly by re- building only the lost RDD partitions.
- Spark can be used to query a 1 TB dataset interactively with latencies of 5–7 seconds

Lately, in the newer versions of Apache Spark, new components have been built and they are now part of the framework. Apache Spark SQL engine is a component that introduced a new API called `DataFrames` (previously called `SchemaRDD`). DataFrames provides support for structured and semi-structured data and the ability to use a SQL like language to be transformed. As in the RDD case, this API evaluates operations lazily so that it can perform relational optimizations. To support the wide range of data sources and algorithms in big data, Spark SQL introduces a novel extensible optimizer called Catalyst. Catalyst makes it easy to add data sources, optimization rules, and data types for multiple domains[Wikimedia, 2017]. The DataFrame API offers rich relational/procedural integration within Spark programs. DataFrames are collections of structured records that can be manipulated using Spark's API, or using new relational APIs that allow richer optimizations. Like an RDD, a DataFrame is an immutable distributed collection of data. Unlike an RDD, data is organized into named columns, like a table in a relational database. Designed to make large data sets processing even easier, DataFrame allows developers to impose a structure onto a distributed collection of data, allowing higher-level abstraction. [Armbrust et al., 2015b]

Starting from Spark 2.0 version, DataFrame APIs have been merged with Datasets APIs, unifying data processing capabilities across libraries. Both DataFrame and Dataset APIs are built on top of the Spark SQL engine, so theyy uses Catalyst to generate an optimized logical and physical query plan. Also, since Spark as a compiler understands Dataset type JVM object, it maps the type-specific JVM object to Tungsten's internal memory representation using Encoders. As a result, Tungsten Encoders can efficiently serialize/deserialize JVM objects as well as generate compact bytecode that can execute at superior speeds [Armbrust et al., 2015b].

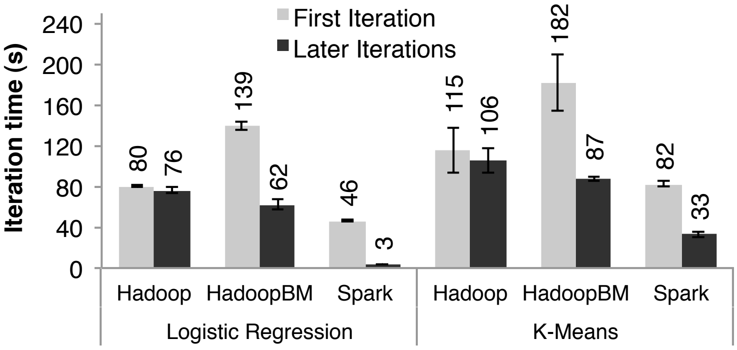


Figure 4.2.: Duration of the first and last itertions in Hadoop, HadoopBinMem and Spark in logistic regression taks and k-means clusterization using 100 GB data in a 100 nodes cluster [Zaharia et al., 2012]

The work of [Shi et al., 2015] highlights the feasibility of the usage of Spark to improve many particular distributed algorithms with different profiles. They performed a quantitative analysis selecting workloads collectively cover the characteristics of typical batch and iterative analytic applications run on MapReduce and Spark. For each type of job, they covered different shuffle selectivity (i.e., the ratio of the map output size to the job input size, which represents the amount of disk and network I/O for a shuffle), job selectivity (i.e., the ratio of the reduce output size to the job input size, which represents the amount of HDFS writes), and iteration selectivity (i.e., the ratio of the output size to the input size for each iteration, which represents the amount of intermediate data exchanged across iterations). For each workload, given the I/O behavior represented by these selectivities, we evaluate its system behavior (e.g., CPU-bound, disk-bound, network-bound) to further identify the architectural differences between MapReduce and Spark.

Overall, they showed that Spark is approximately 2.5x, 5x, and 5x faster than MapReduce, for Word Count, k-means, and PageRank, respectively. Although Spark's performance advantage over MapReduce is known, par- ticularly, they attribute Spark's performance advantage to a number of architectural differences from MapReduce:

- For Word Count and similar workloads, where the map output selectivity can be significantly reduced using a map side combiner, hash-based aggregation in Spark is more efficient than sort-based aggregation in MapReduce. The execution time break-down result indicates that the hash-based framework contributes to about 39% of the overall improvement for Spark.
- For iterative algorithms such as k-means and PageRank, caching the input as RDDs can reduce both CPU (i.e., parsing text to objects) and disk I/O overheads for subsequent iterations. It is noteworthy that the CPU overhead is often the bottleneck in scenarios where subsequent iterations do not use RDD caching. As a result, RDD caching is much more efficient than other low-level caching approaches such as OS buffer caches, and HDFS caching, which can only reduce disk I/O. Through micro- benchmark experiments, we show that reducing parsing (CPU) overhead contributes to more than 90% of the overall speedup for subsequent iterations in k-means.
- Since Spark enables data pipelining within a stage, it avoids materialization overhead for output data on HDFS (i.e., serialization, disk I/O, and network I/O) among iterations for graph analytics algorithms such as PageRank.

An exception to Spark's performance advantage over MapReduce is the Sort workload, for which MapReduce is 2x faster than Spark. This is due to differences in task execution plans. MapReduce can overlap the shuffle stage with the map stage, which effectively hides network overhead which is often a bottleneck for the reduce stage [Shi et al., 2015].

The motivation of this work stems from the possibility of exploiting the advantages of Apache Spark to achieve an implementation of a memory-efficient HDT encoder according to the terms of [Radoi et al., 2014].

## 4.2. Expected contributions

The proposed line of research is to design an algorithm distributed under the MapReduce paradigm with the following considerations:

- Design a speculative triple partitioning strategy that satisfies the new memory usage constraints.

- Add, disaggregate, or balance tasks as much as necessary to minimize the number of writes to disk.
- Study the properties and constraints of the compact structures of the HDT components to design a divide-and-conquer algorithm that allows incremental HDT encoding to be constructed or mergable partitions.
- Perform experiments to measure the performance and memory needs obtained.

On the other hand, with the aim of facilitating reproducibility and guaranteeing the validity of the experiments carried out in the context of this work, a provisioning activity will be carried out to facilitate the task of setting up a cluster environment and run test or validations.

## 4.3. Materials and methodology

For the evaluation of the results of this work, a purely quantitative experimental procedure have been followed in line as those previously performed within the research line. We measured the elapse time in the encoding of data sets of different sizes with detail on the relevant stages to study which ones performs better or worst and why. To study the fine grain behavior of the different technologies, the jobs logs and the detailed information coming from history services (Spark History Server and MapReduce History Server) have been analyzed. All experiments (HDT-MR, and the new HDT-SPARK proposal) have been performed on the same machine with the same configuration of nodes and resources.

The infrastructure used to test the validity and performance of the proposed algorithm and its implementation is based on a virtual environment with dedicated resources. 11 servers have been setup in a master-slave architecture configuration: 1 master, 10 slaves. The master node runs the program driver and spreads, submits and coordinates the tasks for the workers. The table 4.1 shows the virtual servers with resources and the hadoop/spark ecosystem roles they play.

| Server | CPU | RAM | Roles |
|---|---|---|---|
| Master | 4 cores 64bit KVM @2.1GHz | 40GB | HDFS namenode.<br>Spark master.<br>Spark history server.<br>YARN ResourceManager.<br>MapReduce HistoryServer. |
| Slaves 1..10 | 4 cores 64 bit KVM @2.1GHz | 8GB | HDFS datanode.<br>Spark worker.<br>YARN NodeManager. |

Table 4.1.: Hardware configuration of the master and 10 slaves.

The RDF datasets used (see 4.2) are public available in [W3C, 2016], [RDFHDT, 2016] and [LUBM, 2016]. The first 5 dataset are real world data, while latest two are synthetic mashups for test and scalability purposes:

**LUBM 4K** The Lehigh University Benchmark is developed to facilitate the evaluation of Semantic Web repositories in a standard and systematic way. The benchmark is intended to evaluate the performance of those repositories with respect to extensional queries over a large data set that commits to a single realistic ontology. It consists of a university domain ontology, customizable and repeatable synthetic data, a set of test queries, and several performance metrics. The 4K dataset has been generated with the UBA tool ([LUBM, 2016]) for 4000 universities.

**dbpedia+linkedgeodata** The combination of two real world datasets to build a bigger one even though there are no real links among its terms.

| Dataset | #Triples | #Subjects | #Predicates | #Objects | #Shared |
|---|---|---|---|---|---|
| linkedmdb | 3,579,532 | 277,041 | 148 | 1,009,618 | 118,514 |
| geonames | 96,275,507 | 6,417,955 | 28 | 33,236,906 | 125,402 |
| linkedgeodata | 271,180,352 | 10,445,197 | 18,272 | 80,278,063 | 41,471,798 |
| DBPedia | 431,441,103 | 2,779,008 | 58,327 | 86,914,854 | 22,012,722 |
| Freebase | 2,067,068,155 | 2,797,905 | 770,416 | 339,628,695 | 99,203,549 |
| LUBM 4K | 534,203,577 | 86,900,413 | 18 | 64,589,197 | 19,924,941 |
| dbpedia+linkedgeodata | 702,621,455 | 13,224,205 | 76,599 | 167,192,917 | 63,484,520 |

Table 4.2.: Datasets used with triples and components sizes

**Note:** All input datasets are in N-Triples format and compressed with LZO algorithm.

# Part III.

# Developed work

# 5

# Experimental design

The first thing we need to do is to validate the HDT-SPARK implementation to ensure both reference implementations (HDT-MR and HDT-SPARK) generates equivalent HDT files.

As HDT is a lossless compression implementation, the encoding process can be reverted to get the same input data, so we use this principle to ensure the HDT serialization processes are correct. The process to decode HDT back to RDF format has been performed with HDT-Java, the tool developed by the contributors of RDF/HDT project [1]. This will be our gold rule to check the validity of HDT files produced, assuming HDT is correct if we can interpret it with this de facto standard tool.

Based in same input, and same hardware configuration, to measure the performance and resources usage, the following principal KPIs are measured from both implementations:

- Total elapse time.
- The metrics were extracted from the hadoop and spark history services respectively.

## 5.1. Validation procedure

The validation procedure to ensure the output is correct is a three-way comparison performed with hdt-java tool (link) converting the HDT file back to RDF in N-Triples format.

---

[1] http://www.rdfhdt.org

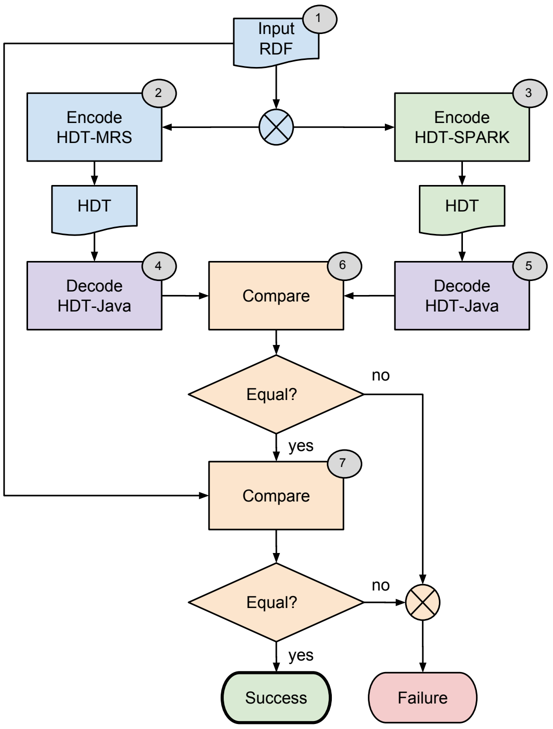


Figure 5.1.: Validation algorithm

- Take input.rdf (in N-Triples format) sorted lexicographically.
- Convert input.rdf to output-hdtmr.hdt with HDT-MR implementation. HDT-MR performance KPIs are extracted at this stage for further analysis.
- Convert input.rdf to output-hdtspark.hdt with HDT-SPARK implementation. HDT-SPARK performance KPIs are extracted at this stage for further analysis.
- Convert output-hdtmr.hdt to output-hdtmr.rdf with HDT-Java tool and sort output RDF lexicographically.
- Convert output-hdtspark.hdt to output-hdtspark.rdf with HDT-Java tool and sort output RDF lexicographically.
- Compare output-hdtmr.rdf against output-hdtspark.rdf
- Final comparison: Compare output-hdtmr.rdf with input.rdf

If final comparison is success. We can ensure the output produced by both implementations are correct and hence, we can be fair comparing the time and resources taken to perform both jobs.

## 5.2. Validation results

Two datasets have been used to validate the HDT-MR and HDT-SPARK results:

**linkedmdb** Linked Data about Movies (3.5M triples. see table 4.2) .

The validation 5.1 was succeeded with few negligible UTF encoding differences comparing with the input. We must assume N-Triples input shall be in a uniform and platform supported encoding like UTF-8 or ASCII, but that's not the case as input has merges of different encodings.

Code 5.1: Sample linkedmdb difference

```
-<http://data.linkedmdb.org/actor/10008> <http://data.linkedmdb.org/movie/actor_name> "V**ra Clouzot" .
+<http://data.linkedmdb.org/actor/10008> <http://data.linkedmdb.org/movie/actor_name> "V\u00C3\u00A9ra Clouzot" .
```

On the other hand, HDT-MR and HDT-SPARK outputs matched perfectly between them.

# 6

# Solution design and implementation

The main objectives in this chapter were to understand how spark works and how we can build a solution for the HDT serialization problem on top of it, analyze the performance of proposed algorithm using the tools and metrics provided by the ecosystem and optimize then to get final approach. To describe the algorithms, we'll use the following visual convention:

**Continue line arrows** DAG dependency which back-propagates inputs.

**Discontinue line arrows** Use inputs but does not back feed the DAG graph.

**Green (olive)** Feeds more than one output, so it should be cached to avoid multiple time (re)computation.

**Wide red frame** Expensive operations requiring shuffle.

Next sections details how HDT-SPARK has been designed and implemented. The problem has been decomposed in three stages:

- Dictionary generation.
- Triples encoding.
- HDT serialization.

## 6.1. Dictionary generation

The first stage parses the input dataset clasifiying the RDF terms by its type:

**subject** (S) If it is the first item in the triple.

**predicate** (P) If it is the second item in the triple.

**object** (O) If it is the third item in the triple.

As one subject term can appear as object term as well, we need to identify one aditional group (SO) having the terms meeting this double role. With the four groups (S, SO, O, P), we are in place to uniquely assign one id to each one. To do that, we'll filter out unique terms within each group, sort them lexicographically to ensure a high reduction ratio in the further compression. Figure 6.1 shows the jobs DAGs with brief sample content.

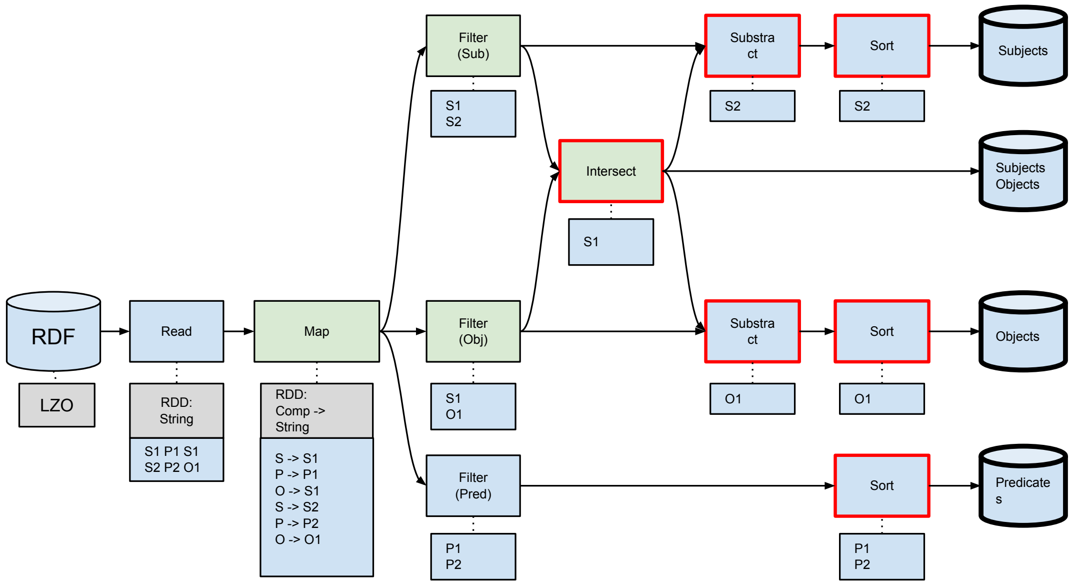


Figure 6.1.: Dictionary generation algorithm

## 6.2. Triple encoding

Triple encoding is the heart of the HDT encoding. It consist on the replace of component literals by unique identifiers. Objective is to save space repeating components and store RDF graph in a succint structure.

Three different implementations have been tested

### 6.2.1. Materialization method

Instead of using any transformation, dictionary dataframes are retrieved to the driver, meaning all the dictionaries are loaded in memory in the Spark main node. This way, the expensive shuffle operations are avoided, but we are in a similar situation than the HDT-MR solution. As we discovered in the first results, this method is suitable for small datasets, but takes more and more memory as long as the dictionary size grows. For that reason, we discarded this approach in the final proposal in order to find a better way to leverage Spark primitives to provide a distribute merge algorithm.

As shown in figure 6.2, Spark DAG is simple tu understand. We start from the ordered dictionaries (subject-object, subject, predicates, objects) which are sorted unique sets wich value starting at 0. To met the HDT specification, we need to med following rules:

- To encode subjects, we need to take the keys from shared dictionary (sujects-objects) and subjects. By definition, both sets keys does not contains common terms, and theirs values starts at 0. In consequence, we add a bias of 1 to the shared dictionary and another bias to the subjects for them to start with the latest shared key plus one. This is performed with two map operations. then a union primitive is used to merge both dictionaries with the keys in the right range.
- To encode the objects, we do the same workflow as in the subjects step to remap the values in the right range.
- The encoding of predicates only needs to add a bias to start by 1.

At this point, we are ready to replace any subject, predicate or objects coming from the input by it's respective key, which is taken from it's transformation (subject, predicate or object). These three datasets are materialized, meaning they are retrieved from the program driver, loaded in memory and then transfered to all workers so they can have a copy in memory. After that, a transformation from the input set (Lookup) replaces the values in a single and fast map operation in parallel from all workers.

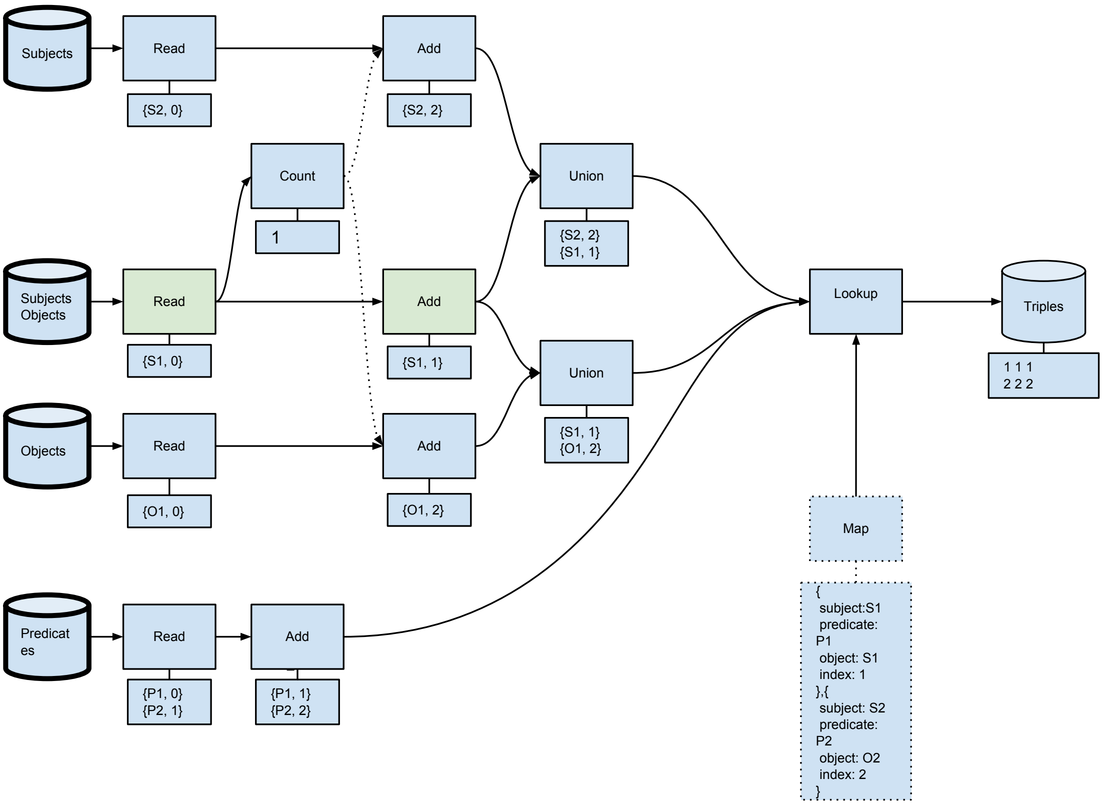


Figure 6.2.: Triples encoding: materialization algorithm

The Spark actions in this workflow are:

- Every single dictionary materialization before lookup.
- The final triples save operation.

All other transformations are lazy. In this case, it's not meaningful to cache or persist any of them because the computation cost to perform them is lower than the effort or memory pressure needed to do the cache, meaning it's best to perform the read and add operation twice on subject-objects than maintain the transformation in memory or disk.

### 6.2.2. Join method

This method (see 6.3) is founded on the usage of the join transformation available in Spark. Basically, the strategy is similar that the previous one, but we've replaced the materialization and distribution of the dictionaries by the join Spark primitive transformation. The Spark Join transformation is similar to the inner join relational database operation, providing a new dataset in the output containing the combination of the records

matching both inputs. In this case, the key to match is the subject, predicate or object component dataframe containing strings:
{`subject_string, predicate_string, object_string`})
to match consecutively the dictionary dataframes
{`subject_string, value_identifier`}
{`predicate_string, value_identifier`}
{`object_string, value_identifier`}
so we can replace the strings by the value identifiers. Thus, the difference comes from the three chained join operations from the three transformed dictionaries. After last join stage, input triples are completely replaces by it's keys in all components.

The unique Spark action in this workflow is the final write of the result. All other transformations are lazily evaluated. The only concern with this method is the cost of the join operation, which leads into expensive shuffle operations to distribute the partitions. Shuffle operations are one of the most relevant bottlenecks we can find in the Spark framework. It means we need to find a balance between data redistribution, causing a high cost in network I/O and other technicals consuming more memory or CPU resources.

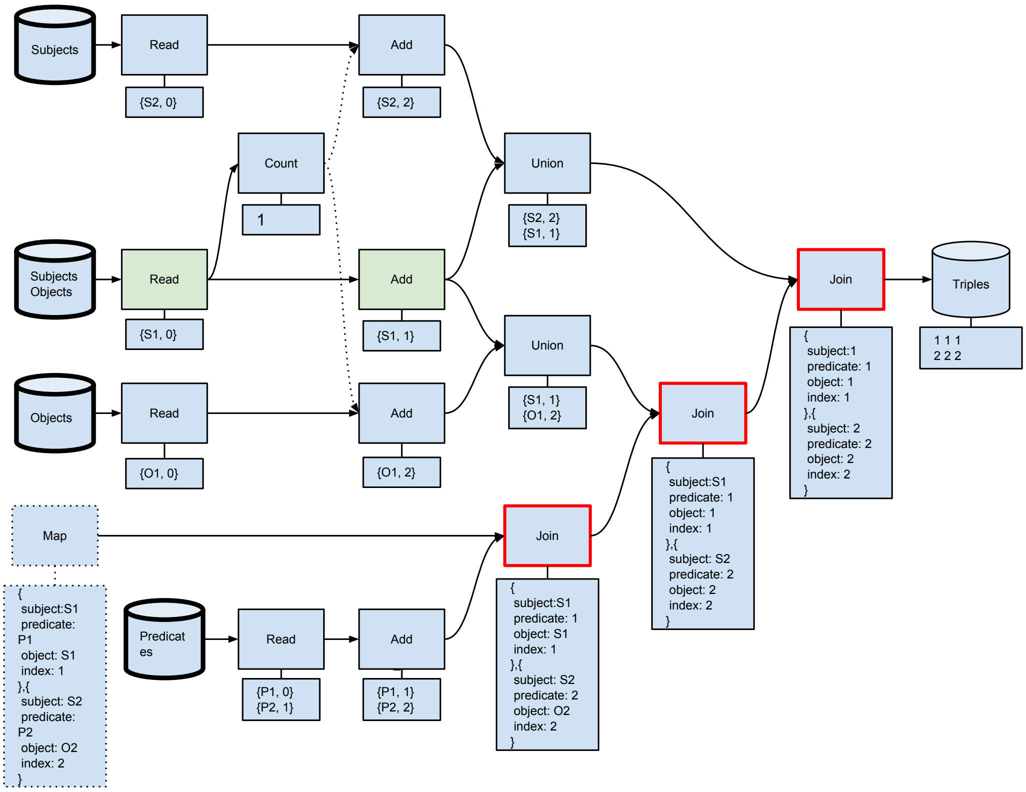

Subjects
Read
{S2, 0}
Add
{S2, 2}
Count
1
Union
{S2, 2}
{S1, 1}
Subjects Objects
Read
{S1, 0}
Add
{S1, 1}
Union
{S1, 1}
{O1, 2}
Join
Triples
1 1 1
2 2 2
{ subject:1 predicate: 1 object: 1 index: 1 },{ subject: 2 predicate: 2 object: 2 index: 2 }
Objects
Read
{O1, 0}
Add
{O1, 2}
Join
{ subject:S1 predicate: 1 object: 1 index: 1 },{ subject: S2 predicate: 2 object: 2 index: 2 }
Map
{ subject:S1 predicate: P1 object: S1 index: 1 },{ subject: S2 predicate: P2 object: O2 index: 2 }
Join
{ subject:S1 predicate: 1 object: S1 index: 1 },{ subject: S2 predicate: 2 object: O2 index: 2 }
Predicates
Read
{P1, 0}
{P2, 1}
Add
{P1, 1}
{P2, 2}

Figure 6.3.: Triples encoding: join algorithm

### 6.2.3. Cogroup method

As Join method introduces expensive shuffle operations, we tried different approaches to resolve the triples encoding problem. Cogroup is a Spark transformation primitive more efficient than the join in terms of task distribution. As it can be shown in figure 6.4, it's nearly equivalent with few adds which have been simplified in the workflow. Cogroup transformation groups two KeyValue RDDs (PairRDD). For instance Consider, we have two RDDs of {`Subject,V1`} and {`Subject,V2`} types, after transformation is executed, result will be a RDD with the common key and the list of values in a tuple: {`Subject,(Iterable<V1>,Iterable<V2>`} type.

Concretely, the algorithm consist on two stages prior the cogroup show in figure 6.4:

1. cogroup every dictionary value to the row number in which it appears.
2. unfold the cogrouped {`Subject, Iterable<Value>, Iterable<RowNum>`} with flatMap primitive to get a PairRDD of {`RowNum, Key`}

We call this lazy preprocessing stage ”row encoding”. Then we consecutively cogroup this three transformations with the input as {`RowNum, Triple`}. As there can be only one coincidence because RowNums are unique by row, we can just take the first element in the iterable as the resolved identifier.

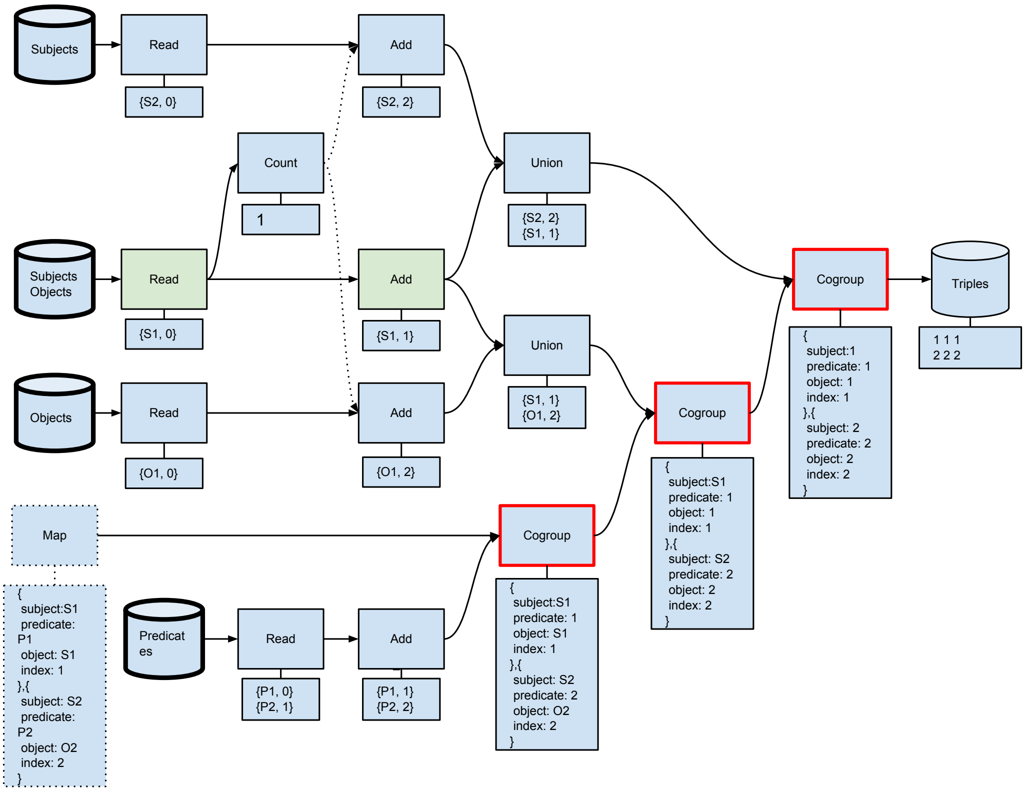

Subjects
Read
{S2, 0}
Add
{S2, 2}
Count
1
Union
{S2, 2}
{S1, 1}
Subjects
Objects
Read
{S1, 0}
Add
{S1, 1}
Union
{S1, 1}
{O1, 2}
Objects
Read
{O1, 0}
Add
{O1, 2}
Cogroup
Triples
1 1 1
2 2 2
{ subject:1 predicate: 1 object: 1 index: 1 },{ subject: 2 predicate: 2 object: 2 index: 2 }
Cogroup
{ subject:S1 predicate: 1 object: 1 index: 1 },{ subject: S2 predicate: 2 object: 2 index: 2 }
Map
{ subject:S1 predicate: P1 object: S1 index: 1 },{ subject: S2 predicate: P2 object: O2 index: 2 }
Cogroup
{ subject:S1 predicate: 1 object: S1 index: 1 },{ subject: S2 predicate: 2 object: O2 index: 2 }
Predicates
Read
{P1, 0}
{P2, 1}
Add
{P1, 1}
{P2, 2}

Figure 6.4.: Triples encoding: cogroup algorithm

## 6.3. First results and further optimizations

The first goal we needed to achieve was to settle on which triples encoding approach to follow. Even though the materialization method is simple and effective, it comes with a very big memory constraint, so we discarded it.

Cogroup method probed to be more efficient in small datasets but it had some disadvantages:

- Arbitrary big memory consumption depending on the number of repetitions of one component in many rows than can stress the memory available to process a single RDD partition.

- Relies on RDD processing, which is more CPU and memory intensive compared to dataframes. That becomes more and more visible the bigger a dataset is.

For these reasons, we've also discarded this approach in favor of join transformation using dataframes.

Next thing we had to do to really improve the new algorithm was to analyze the graph to identify duplicate work and orchestrate every single and independent operation to run in parallel, so we can wait only for the stages from we need to consume. This resulted in a 4 step algorithm in which we:

1. Index the input data (lzo text format).
2. Build all the interim dictionaries in parallel and get some statistics we need to build the header, as the number of components and maximum string lengths.
3. Given previous stage and statistics, we encode the triples following the HDT specification. Mean while, the header and dictionary is also transfered to the driver at the master node and serialized in HDT.
4. Once triples are encoded, we transfer them to the driver at the master node where HDT serialization is completed.

Figure 6.5 shows the final execution model of the HDT-SPARK proposal.

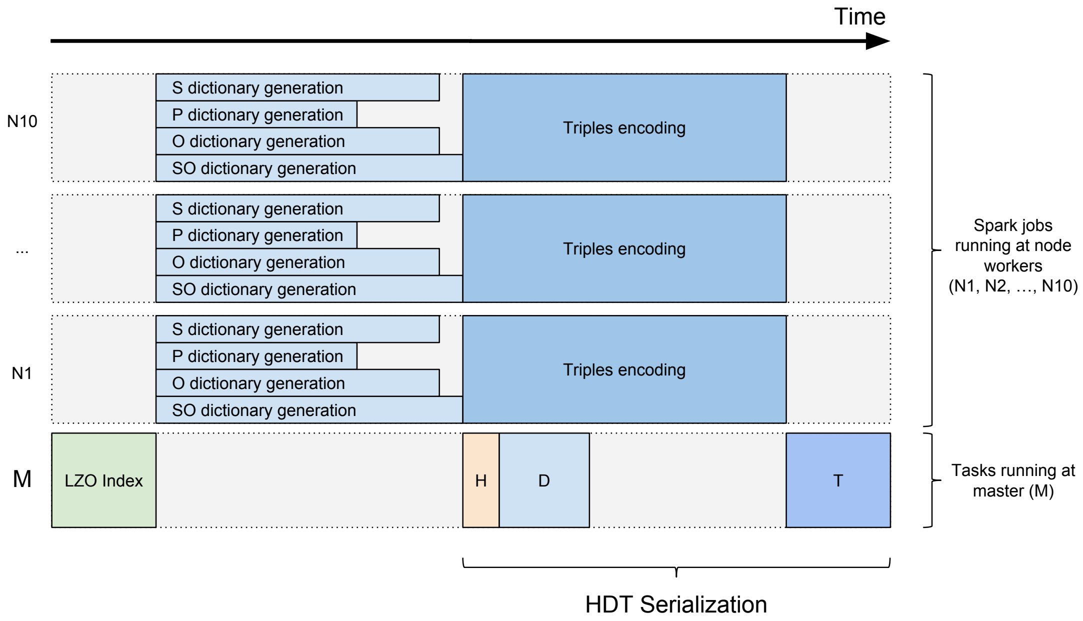


Figure 6.5.: HDT-SPARK execution model

## 6.4. Lessons learned

Based on the inputs found in the literature and the first stress tests we performed,

- Spark dataframes are in general the desirable API on which to base the developments. They are easy to use, faster and consumes less memory than RDD. However, that does not means RDD are not necessary. They are, so many primitives are only available in the RDD API. Also, RDDs are very helpful writing transformations and filters in Java code.
- Not all operations preserve dataset order. Even if they does not shuffle data, like `coallesce` or `repartition` with shuffle flag unset.
- There is not fixed rules to optimize a problem. For instance, sometimes altering the `order` and `distinct` operations can produce different performance results. One specific order can be faster for small dataset but slower with big ones.
- Best way to understand how the Apache Spark works is to study the history server timings and execution plans.

- Infrastructure services provisioning and roles distributions across nodes are not straightforward matters and might cause good ideas to not succeed if wrong decisions are made.

# 7

# Experiment results

In this chapter we discuss the outcomes that have been achieved with the HDT-SPARK implementation and how technology can be used to improve it.

## 7.1. Results

The test performed with different dataset sizes showed the benefits obtained with the proposed solution compared to the implementation of Hadoop MapReduce, as well as some highlights to further improve the algorithm.

Two types of datases have been used (see table 4.2):

**Real world datasets** 5 RDF datasets with different triples sizes. In growing order: linkedmdb (3.5M), geonames (96M), linkedgeodata (271M), DBPedia (431M), Freebase(2067M).

**Synthetic/mashups** Synthetically produced datasets (LUBM-4K, from the Lehigh University Benchmark), or manual merge of different datasets to become larger (dbpedia+linkedgeodata).

Both types are useful because even though nowadays we can find very large RDF datasets to analyze performance or limitations on the processing algorithms, synthetic or merges lets us analyze how they perform with big amounts of different components or some uncommon situations or border limits.

Tables 7.1 and 7.2 shows the total and per-stage elapsed time in seconds to process target datasets in N-Triples format compressed with LZO with both implementations in the same hadoop cluster. Stages correspond to the different algorithm steps described on section 6.3:

**Total** Total elapsed time in seconds to build the HDT dataset.

**Index** Time in seconds to build the input lzo index.

**Dictionary** Time in seconds to build the dictionary.

**Triples** Time in seconds to encode the triples.

**Finalization** Time in seconds to finalize the HDT serialization (read inputs, write sections, compute CRC...).

| | **HDT-MR elapsed time (s)** | | | | |
|---|---|---|---|---|---|
| **Dataset** | **Total** | **Index** | **Dictionary** | **Triples** | **Finalization** |
| linkedmdb | 214.75 | 0.91 | 84.00 | 114.00 | 15.84 |
| geonames | 1163.26 | 24.69 | 346.00 | 589.00 | 203.573 |
| linkedgeodata | 3414.19 | 39.93 | 689.00 | 1928.00 | 757.26 |
| DBPedia | 7723.51 | 85.69 | 2716.00 | 4238.00 | 683.82 |
| Freebase | Unable to process by out of memory | | | | |
| LUBM 4K | 3866.31 | 132.40 | 829.00 | 2354.00 | 550.91 |
| dbpedia+linkedgeodata | 15750.70 | 136.47 | 4451.00 | 9779.00 | 1384.23 |

Table 7.1.: Elapsed time to process RDF nt+lzo datasets with HDT-MR

| | **HDT-SPARK elapsed time (s)** | | | | |
|---|---|---|---|---|---|
| **Dataset** | **Total** | **Index** | **Dictionary** | **Triples** | **Finalization** |
| linkedmdb | 129.47 | 0.64 | 57.90 | 56.03 | 14.88 |
| geonames | 693.60 | 14.10 | 295.38 | 281.63 | 102.49 |
| linkedgeodata | 1588.70 | 41.53 | 714.77 | 553.98 | 278.40 |
| DBPedia | 2052.79 | 77.72 | 901.12 | 653.67 | 420.27 |
| Freebase | 9758.58 | 330.03 | 3693.81 | 3673.23 | 2061.50 |
| LUBM 4K | 2081.23 | 43.77 | 961.77 | 752.14 | 323.53 |
| dbpedia+linkedgeodata | 3514.74 | 115.99 | 1526.81 | 1172.31 | 699.63 |

Table 7.2.: Elapsed time to process RDF nt+lzo datasets with HDT-SPARK

**Note:** Measured times remained stable with low variance along 3 consecutive repetitions (less than 10% deviation).

## 7.2. Discussion

First singular situation we observe is that HDT-MR was not able to compute Freebase. Reason is this dataset has a big number of very different components among them. That situation caused Plain Front Coding algorithm to not be able to achieve big enough compression ratio and hence HDT-MR tried to allocate more than the 6GB of RAM memory available in workers (2 GB remaining are reserved for services, file caching and Operating System). For that reason, even though HDT-MR is able to scale to compute very big datasets as it has been proved before, there exists this memory limitation which is not such often when computing synthetic RDF datasets, as most of the components are common and Plain Front Coding gets a big compression ratio there.

The performance overall, as expected is higher in HDT-SPARK. As summarized in table 7.3, HDT-SPARK is between 1.7 and 2.1 times faster in normal cases. There are some exceptions like DBPedia and DBPedia+linkedgeodata in which performace is even greater.

| Dataset | Improvement |
|---|---:|
| linkedmdb | 1.7 x |
| geonames | 1.7 x |
| linkedgeodata | 2.1 x |
| DBPedia | 3.8 x |
| Freebase | - |
| LUBM 4K | 1.9 x |
| dbpedia+linkedgeodata | 4.5 x |

Table 7.3.: Performance of HDT-SPARK over HDT-MR (times faster)

This two cases are singular because the resulting dictionaries are very big in comparison with others. According to the table 7.4 showing HDT output sizes, we can see the size is anomaly big compared with others having more triples and components. That means the dictionaries are big because it contains many components without common prefix and seems to cause a degradation in the HDT-MR algorithm.

| Dataset | HDT size |
|---|---|
| linkedmdb | 32 MB |
| geonames | 860 MB |
| linkedgeodata | 6.5 GB |
| DBPedia | 6.5 GB |
| Freebase | 26 GB |
| LUBM 4K | 3.1 GB |
| dbpedia+linkedgeodata | 13 GB |

Table 7.4.: HDT Sizes

Figures 7.1 and 7.2 shows graphically the per-stage elapsed time comparison between the two implementations:

**Index** There is no relevant difference in this stage, as it is exactly the same.

**Dictionary** Except in DBPedia and DBPedia+linkedgeodata, Dictionary stage is nearly equal or slightly faster in HDT-MR. This is a well know sort performance situation highlighted by [Shi et al., 2015] summarized in section 4.1.

**Triples** Very big performance gain in HDT-SPARK. More than 2x in worst cases. Unfortunately, top performance is limited by the need have sort stages, which is, in general one of the bottlenecks in this paradigm.

**Finalization** In general, HDT-MR performs better in this stage as it uses custom binary formats. However, as HDT-SPARK is able to serialize in a parallel pipeline, the bad effect is isolated with a big gain at the end (2x).

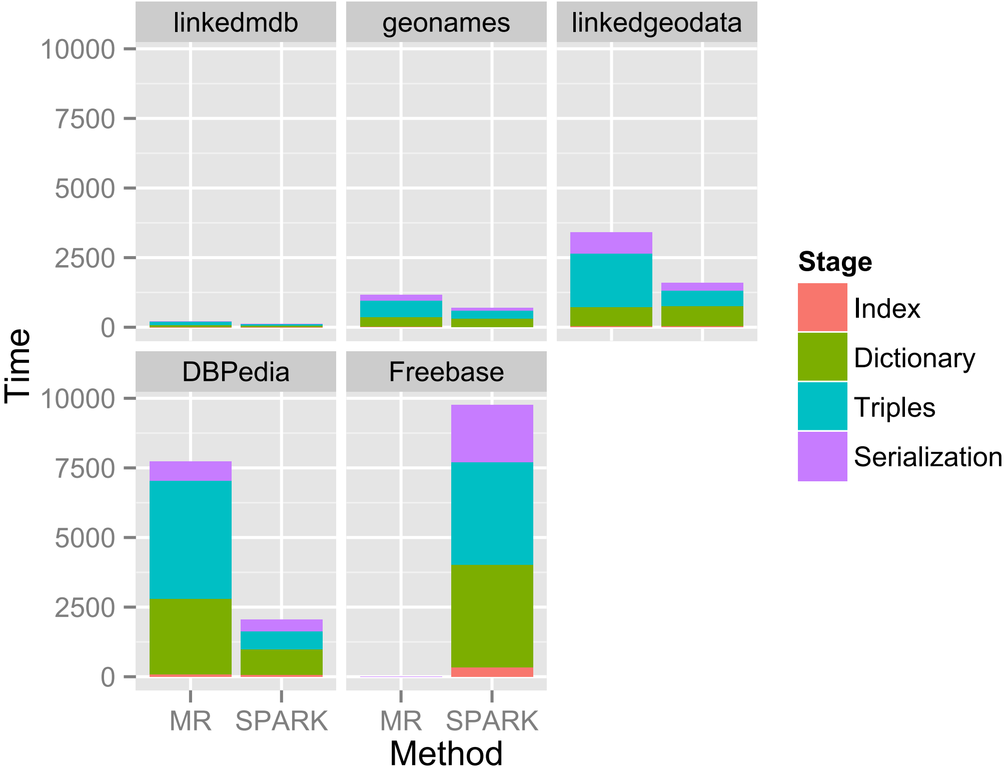


Figure 7.1.: HDT-MR and HDT-SPARK performance comparison on real data input datasets

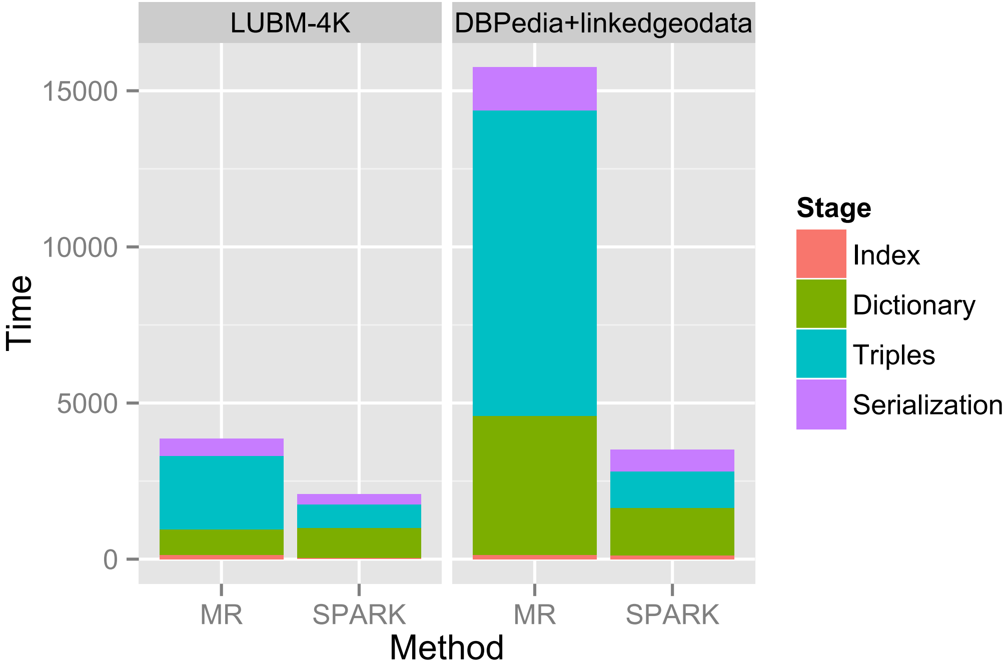


Figure 7.2.: HDT-MR and HDT-SPARK performance comparison on synthetic/mashup datasets

# Part IV.

# Finalization

# 8

# Related work

A focused search in some of the most popular scientific publications with the keywords “Apache Spark”, “HDT” and “RDF” shows several similar initiatives to leverage Apache Spark to outperform SPARQL queries over RDF datasets:

- “Presto-RDF: SPARQL Querying over Big RDF Data” [Mammo and Bansal, 2015].
- “SPARQL query processing with Apache Spark” [Naacke et al., 2016].
- “SPARQLGX in Action: Efficient Distributed Evaluation of SPARQL with Apache Spark” [Graux et al., 2016].

They are different approaches willing to provide on-line solutions, meaning the queries are performed on real time while the goal in this research line is to provide off-line processing on commodity hardware.
We found also other lines trying to use different technologies to solve the RDF problem:

- “TriAD: a distributed shared-nothing RDF engine based on asynchronous message passing” [Gurajada et al., 2014].

Furthermore, increasing number of data is currently published adopting Apache Spark to solve other problems:

- “Matrix Computations and Optimization in Apache Spark” [Bosagh Zadeh et al., 2016].
- “Balanced Graph Partitioning with Apache Spark” [Carlini et al., 2014].
- “Un enfoque MapReduce del algoritmo k-vecinos más cercanos para Big Data” [Maillo et al., 2003].
- “SparkSeq: fast, scalable and cloud-ready tool for the interactive genomic data analysis with nucleotide precision” [Wiewiórka et al., 2014].

# 9

# Conclusion

Given the results, Apache Spark is a good technology to set the bases to build better HDT serialization implementations. Even though there is not a big gain in all cases in the dictionary generation stage, there are some benefits which makes it a good choice:

- Most important achievement is we remove the memory requirement in the workers to compute a dataset. With HDT-SPARK, the dictionaries are not fully loaded in memory or transfered to the workers in the triples encoding stage. As we have demonstrated, we can build an arbitrary large dataset without theoretical memory constrains in workers, as we was able to process a very big 2Billion triples RDF dataset for which HDT-MR failed in the same environment with same resources as HDT-MR was not able to hold very big dictionaries.
- HDT-SPARK outperformed between 1.5x and 1.9x in medium sized RDF datasets and over 3.0x faster for some big real world datasets.
- The dictionary stage, even if it's not much faster than MapReduce approach, is important because it sets the inputs of further stages in a standard format ready to use not only by spark, but any other technology from the hadoop ecosystem.
- In the HDT-SPARK approach, the less something depends on one stage output, the more it can be parallelized. This lets us to fine-grain control the task we run in parallel.

To met the reproducibility secondary objective and facilitate the continuation of future work, a software provisioning has been organized for anyone to be able to deploy the cluster distribution in a virtual machine (or cluster of virtual machines) or docker containers.

# 10

# Limitations and future work

Main bottleneck still persisting after this work is the need of fetching all the dictionary partitions and encoded triples back to the driver and serialize the HDT in the master node. New questions come to the fore, like how to build a complete end to end parallel HDT serialization. Unfortunately, to do this, we need to rethink how to split and merge back the succint structures without compromise.

These outcomes open up interesting new lines of research like:

- Discovery of new dictionary and triples compact data structures with better properties suitable able to be partitioned.
- Improve current solution with the design of a new algorithm to serialize Spark partitions of dictionary or triples bitmap (RDD or dataset) to eliminate the need of a final single serialization.
- Improve HDT for the triples section to be decomposed in ordered chunks. This will let us solve the serialization bottleneck, but will break the compatibility, so SPARQL queries must be repeated on all triples chunks. However, many optimizations are possible.
- Study the computation effort and cost/gain of adding extra representations or indexes in HDT to allow efficient SPARQL queries by predicate, by object and/or combined.

The reduction of the computation effort to serialize RDF is a very important milestone to achieve, because Hopefully, new alternative ways to leverage the technology to speedup the HDT serialization process like applying a GPGPU-based implementation like [Choksuchat and Chantrapornchai, 2013] within Spark makes sense, and according to the literature, hybrid GPGPU and MPP approaches are not yet explored in deep.

# Bibliography

[Arias Gallego et al., 2011] Arias Gallego, M., Fernández, J. D., Martınez-Prieto, M. A., and Gutierrez, C. (2011). HDT-it: Storing, Sharing and Visualizing Huge RDF Datasets. In *10th International Semantic Web Conference, Bonn, Germany, October*, pages 23–27.

[Armbrust et al., 2015a] Armbrust, M., Das, T., Davidson, A., Ghodsi, A., Or, A., Rosen, J., Stoica, I., Wendell, P., Xin, R., and Zaharia, M. (2015a). Scaling Spark in the Real World: Performance and Usability. *Proc. VLDB Endow.*, 8(12):1840–1843.

[Armbrust et al., 2015b] Armbrust, M., Xin, R. S., Lian, C., Huai, Y., Liu, D., Bradley, J. K., Meng, X., Kaftan, T., Franklin, M. J., Ghodsi, A., and Zaharia, M. (2015b). Spark SQL: Relational Data Processing in Spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1383–1394, New York, NY, USA. ACM.

[Berners-Lee, 2006] Berners-Lee, T. (2006). Linked Data - Design issues `https://www.w3.org/DesignIssues/LinkedData.html` [last visit: Aug 2017].

[Bizer et al., 2009] Bizer, C., Heath, T., and Berners-Lee, T. (2009). Linked data-the story so far. *Semantic Services, Interoperability and Web Applications: Emerging Concepts*, pages 205–227.

[Bosagh Zadeh et al., 2016] Bosagh Zadeh, R., Meng, X., Ulanov, A., Yavuz, B., Pu, L., Venkataraman, S., Sparks, E., Staple, A., and Zaharia, M. (2016). Matrix computations and optimization in apache spark. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 31–38. ACM.

[Carlini et al., 2014] Carlini, E., Dazzi, P., Esposito, A., Lulli, A., and Ricci, L. (2014). Balanced graph partitioning with Apache Spark. In *European Conference on Parallel Processing*, pages 129–140. Springer.

[Choksuchat and Chantrapornchai, 2013] Choksuchat, C. and Chantrapornchai, C. (2013). On the HDT with the Tree Representation for Large RDFs on GPU. In *Proceedings of the 2013 International Conference on Parallel and Distributed Systems*, ICPADS '13, pages 651–656, Washington, DC, USA. IEEE Computer Society.

[Fernández et al., 2011] Fernández, J. D., Martínez-Prieto, M. A., Arias, M., Gutierrez, C., Álvarez-García, S., and Brisaboa, N. R. (2011). Lightweighting the Web of Data Through Compact RDF/HDT. In *Proceedings of the 14th International Conference on Advances in Artificial Intelligence: Spanish Association for Artificial Intelligence*, CAEPIA'11, pages 483–493, Berlin, Heidelberg. Springer-Verlag.

[Fernández et al., 2010] Fernández, J. D., Martínez-Prieto, M. A., and Gutierrez, C. (2010). Compact representation of large RDF data sets for publishing and exchange. In *The Semantic Web-ISWC 2010*, pages 193–208. Springer.

[Fernández et al., 2013] Fernández, J. D., Martínez-Prieto, M. A., Gutiérrez, C., Polleres, A., and Arias, M. (2013). Binary RDF Representation for Publication and Exchange (HDT). *Web Semantics: Science, Services and Agents on the World Wide Web*, 19:22–41.

[Giménez, 2014] Giménez, J. M. (2014). HDT-MR : A Scalable Solution for RDF Compression with HDT and MapReduce. In *The Semantic Web. Latest Advances and New Domains*, pages 253–268. Springer.

[Graux et al., 2016] Graux, D., Jachiet, L., Genevès, P., and Layaïda, N. (2016). SPARQLGX in Action: Efficient Distributed Evaluation of SPARQL with Apache Spark. In *15th International Semantic Web Conference*.

[Gurajada et al., 2014] Gurajada, S., Seufert, S., Miliaraki, I., and Theobald, M. (2014). TriAD: a distributed shared-nothing RDF engine based on asynchronous message passing. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 289–300. ACM.

[Hernández Illera et al., 2015] Hernández Illera, A., Martínez Prieto, M. A., and Fernández, J. D. (2015). Serializing RDF in Compressed Space. *Data Compression Conference (DCC), Snowbird, UT*, pages 363–372.

[LUBM, 2016] LUBM (2016). The Lehigh University Benchmark (LUBM).

[Maillo et al., 2003] Maillo, J., Triguero, I., and Herrera, F. (2003). Un enfoque MapReduce del algoritmo k-vecinos más cercanos para Big Data. *ACM*, 7:971–980.

[Mammo and Bansal, 2015] Mammo, M. and Bansal, S. K. (2015). Presto-RDF: SPARQL querying over big rdf data. In *Australasian Database Conference*, pages 281–293. Springer.

[Martínez-Prieto et al., 2012] Martínez-Prieto, M. A., Arias Gallego, M., and Fernández, J. D. (2012). Exchange and Consumption of Huge RDF Data. In *Proceedings of the 9th International Conference on The Semantic Web: Research and Applications*, ESWC'12, pages 437–452, Berlin, Heidelberg. Springer-Verlag.

[Naacke et al., 2016] Naacke, H., Curé, O., and Amann, B. (2016). SPARQL query processing with Apache Spark. *arXiv preprint arXiv:1604.08903*.

[Pérez et al., 2006] Pérez, J., Arenas, M., and Gutierrez, C. (2006). Semantics and Complexity of SPARQL. In *International Semantic Web Conference*, volume 4273, pages 30–43. Springer.

[Quilitz and Leser, 2008] Quilitz, B. and Leser, U. (2008). *Querying distributed RDF data sources with SPARQL*, chapter The Semantic Web: Research and Applications: 5th European Semantic Web Conference, ESWC 2008, Tenerife, Canary Islands, Spain, June 1-5, 2008 Proceedings, pages 524–538. Springer.

[Radoi et al., 2014] Radoi, C., Fink, S. J., Rabbah, R., and Sridharan, M. (2014). Translating Imperative Code to MapReduce. *SIGPLAN Not.*, 49(10):909–927.

[RDFHDT, 2016] RDFHDT (2016). Most useful/popular datasets from the LOD cloud in HDT format.

[Schmachtenberg et al., 2014] Schmachtenberg, M., Bizer, C., and Paulheim, H. (2014). State of the LOD Cloud 2014 `http://linkeddatacatalog.dws.informatik.uni-mannheim.de/state/` [last visit: Aug 2017].

[Shi et al., 2015] Shi, J., Qiu, Y., Minhas, U. F., Jiao, L., and Wang, C. (2015). Clash of the Titans: MapReduce vs . Spark for Large Scale Data Analytics. *VLDB*, 8(3):2110–2121.

[W3C, 2014a] W3C (2014a). Notation3 (N3): A readable RDF syntax.

[W3C, 2014b] W3C (2014b). RDF 1.1 Concepts and Abstract Syntax.

[W3C, 2014c] W3C (2014c). RDF 1.1 N-Quads. A line-based syntax for RDF datasets.

[W3C, 2014d] W3C (2014d). RDF 1.1 N-Triples. A line-based syntax for an RDF graph.

[W3C, 2014e] W3C (2014e). RDF 1.1 Turtle. Terse RDF Triple Language.

[W3C, 2016] W3C (2016). Linked datasets available as RDF dumps.

[Wiewiórka et al., 2014] Wiewiórka, M. S., Messina, A., Pacholewska, A., Maffioletti, S., Gawrysiak, P., and Okoniewski, M. J. (2014). SparkSeq: fast, scalable and cloud-ready tool for the interactive genomic data analysis with nucleotide precision. *Bioinformatics*, 30(18):2652–2653.

[Wikimedia, 2017] Wikimedia (2017). Apache spark `https://en.wikipedia.org/wiki/Apache_Spark` [last visit: Aug 2017].

[Wikimedia, nd] Wikimedia (n.d.). Linked data `https://en.wikipedia.org/wiki/Linked_data` [last visit: Aug 2017].

[Yu, 2014a] Yu, L. (2014a). *A Developer's Guide to the Semantic Web*, chapter Linked Open Data, pages 415–473. Springer Berlin Heidelberg, Berlin, Heidelberg.

[Yu, 2014b] Yu, L. (2014b). *A Developer's Guide to the Semantic Web*, chapter The Building Block for the Semantic Web: RDF, pages 23–95. Springer Berlin Heidelberg, Berlin, Heidelberg.

[Yu, 2014c] Yu, L. (2014c). *A Developer's Guide to the Semantic Web*, chapter SPARQL: Querying the Semantic Web, pages 265–353. Springer Berlin Heidelberg, Berlin, Heidelberg.

[Zaharia et al., 2012] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M. J., Shenker, S., and Stoica, I. (2012). Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association.

[Zaharia et al., 2010] Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., and Stoica, I. (2010). Spark: Cluster Computing with Working Sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, volume 10, page 10.

[Zaharia et al., 2008] Zaharia, M., Konwinski, A., Joseph, A. D., Katz, R. H., and Stoica, I. (2008). Improving mapreduce performance in heterogeneous environments. In *OSDI*, volume 8, page 7.